Recoverable FCFS Mutual Exclusion with Wait-Free Recovery

Prasad Jayanti[*1] **and Anup Joshi**[†2]

1 **6211 Sudikoff Lab for Computer Science, Dartmouth College, Hanover, NH, USA**
prasad@cs.dartmouth.edu

2 **6211 Sudikoff Lab for Computer Science, Dartmouth College, Hanover, NH, USA**
anupj@cs.dartmouth.edu

Abstract

Traditional mutual exclusion locks are not resilient to failures: if there is a power outage, the memory is wiped out. Thus, when the system comes back on, the lock will have to be restored to the initial state, i.e., all processes are rolled back to the Remainder section and all variables are reset to their initial values. Recently, Golab and Ramaraju showed that we can improve this state of the art by exploiting the Non-Volatile RAM (NVRAM). They designed algorithms that, by maintaining shared variables in NVRAM, allow processes to recover from crashes on their own without a need for a global reset, even though a crash can wipe out the local memory of a process.

We present a Recoverable Mutual Exclusion algorithm using the commonly supported CAS primitive. The main features of our algorithm are that it satisfies FCFS, it ensures that each process recovers in a wait-free manner, and in the absence of failures, it guarantees a worst-case Remote Memory Reference (RMR) complexity of $O(\lg n)$ on both Cache Coherent (CC) and Distributed Shared Memory (DSM) machines, where n is the number of processes for which the algorithm is designed. This bound matches the $\Omega(\lg n)$ RMR lower bound by Attiya, Hendler, and Woelfel for Mutual Exclusion algorithms that use comparison primitives.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases concurrent algorithm, synchronization, mutual exclusion, recovery, fault tolerance, non-volatile main memory, shared memory, multi-core algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.30

1 Introduction

Recent research has focused on exploiting non-volatile main memory to design algorithms that can tolerate process crashes. The underlying idea of these algorithms is that upon a crash a process executes a recovery code that consults the shared state stored in the non-volatile main memory and brings the system back to a usable state. Due to an explicit recovery procedure to emerge out of a crash, these algorithms are called *recoverable* algorithms. We present a recoverable algorithm for the *FCFS Mutual Exclusion* problem.

* The first author is grateful for the generous support of James Frank Family Professorship.
† The second author is grateful for the support of Dartmouth Fellowship.

31st International Symposium on Distributed Computing (DISC 2017).
Editor: Andréa W. Richa; Article No. 30; pp. 30:1–30:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Let us recall the traditional (non-recoverable) mutual exclusion problem [2], which captures the requirements of a system of processes that share an exclusive resource – a resource that can be accessed by only one process at a time. Specifically, we consider a system consisting of $n > 1$ asynchronous processes, where each process cycles through four sections of code – Remainder, Try, Critical Section (CS), and Exit. Initially, all processes are in the Remainder section. When a process gets interested in accessing the resource, it executes the Try section, where it competes with other processes for exclusive access to the resource. When the Try section terminates, the process moves to CS, where it actually accesses the resource. When the process no longer needs the resource, it executes the Exit section to relinquish its access to the resource. A standard version of the *mutual exclusion problem* is to come up with code for the Try and Exit sections so that the following properties are ensured:

(i) *mutual exclusion:* at most one process is in the CS at any time;

(ii) *bounded exit:* every process completes the exit section in a bounded number of its steps; and

(iii) *starvation freedom:* if a process is in the Try section, it eventually enters the CS (under the assumptions that every process that enters the CS eventually leaves the CS, and no process permanently stops taking steps while in the Try or Exit sections).

We could additionally require the fairness property FCFS [8] which, informally, states that processes enter the CS in the order in which they request the CS. A *mutual exclusion lock* is a solution to the mutual exclusion problem.

A traditional mutual exclusion lock cannot gracefully tolerate failures. In particular, if a process crashes, its memory – including its Program Counter – can lose their contents. As a result, when a process restarts after a crash, it cannot tell where in the algorithm it had failed and from where and how to resume its execution. Thus, process crashes can render a lock permanently unusable. One solution would be a global reset: to recover from a crash, roll back all processes to the Remainder section and set all variables to their initial values. There are two disadvantages to such a global reset. First, if the crash occurs while a process is in the CS and while the data structures manipulated in the CS are in an inconsistent state, the inconsistency will persist even after the reset. Second, a local failure (e.g., the crash of a single process) causes the reset of the entire system, which can be unacceptable in very large systems where the probability that some process or the other crashes is non-negligible.

It would be ideal if the lock can be designed so that the failure of some processes do not affect other processes and even the failed processes, when they eventually restart, can proceed as if they had never failed. Golab and Ramaraju's recoverable mutual exclusion algorithm [4] shed some light on how one might realize this goal by cleverly exploiting hybrid architectures [10] that use NVRAM (nonvolatile random access memory) technology for shared memory. When a process crashes, all its local variables are wiped out. So, when a process restarts after a crash, the general idea for an algorithm is that the process could consult the non-volatile shared memory to reconstruct its lost state and then resume its execution of the algorithm from the point where it crashed, making it seem like the crash never occurred. For instance, if a process crashes while in the CS, after restart a recovery procedure could put the process right back in the CS, making the crash appear simply like a long delay.

Ensuring such seamless recovery while also guaranteeing efficiency is where the intellectual challenge lies. At first glance it might seem that if each process stores its state in the non-volatile memory after every step, then when restarting after a crash, the process should be able to easily restore its state, thereby rendering the design of a recoverable algorithm easy. This strategy however falls short because, even with such a laborious storing of its state, when a process attempts to restore its state after a crash, if the stored value for its

program counter is k, it cannot distinguish whether it had crashed before or after executing the instruction at k. This seemingly small ambiguity makes the design of an algorithm challenging. To appreciate this fact, a reader familiar with the MCS lock should imagine the plight of a recovering process that crashed right after the initial swap operation.

We present an implementation of a recoverable lock using CAS that has the following merits:

(i) it satisfies FCFS,

(ii) its recovery is wait-free, and

(iii) it has a Remote Memory Reference (RMR) complexity of $O(f + \lg n)$ on both Cache Coherent (CC) and Distributed Shared Memory (DSM) machines, where n is the number of processes for which the lock is designed and f is the number of times a process crashes between the time it invokes and exits the lock.

The wait-free recovery of our algorithm ensures that when a process that crashes in the CS subsequently restarts, it gets back into the CS in a bounded number of its own steps, i.e., without any waiting whatsoever, regardless of however many other processes have failed or are slow. Attiya, Hendler, and Woelfel proved a lower bound that the RMR complexity is $\Omega(\lg n)$ for even a non-recoverable CAS-based lock [1]. Thus, our algorithm adds only $O(1)$ RMRs per crash. It would be interesting to resolve if the $O(f + \lg n)$ complexity of our algorithm is optimal.

In comparison, Golab and Ramaraju's locks [4] do not satisfy FCFS and, except for their tournament based algorithm, they do not have wait-free recovery. Ramaraju's lock [11] satisfies FCFS, but it works only on CC machines and uses the memory-to-memory swap primitive, which is not supported on any of the current multiprocessors. Golab and Hendler adapt the MCS lock in three different ways to implement recoverable locks. Two of their locks, although not claimed in their paper, appear to satisfy FCFS. One of these two works only on CC machines; further, even a process that crashes just once can incur $\Omega(n)$ RMRs in the worst case (but on the positive side a process that does not crash incurs at most $O(1)$ RMRs). The other algorithm of theirs that satisfies FCFS uses a double word primitive, which is not supported on any of the current multiprocessors.

Recoverable algorithms are typically designed by adapting existing (non-recoverable) mutual exclusion algorithms. Golab and Ramaraju's recoverable algorithms [4] adapt Yang and Anderson's tournament algorithm [12]. Ramaraju's [11] and Golab and Hendler's recoverable algorithms [3] adapt Mellor-Crummey and Scott's algorithm [9]. Our algorithm in this paper adapts Jayanti's mutual exclusion algorithm [7].

2 Model and Specification

The system consists of n processes named $1, 2, \ldots, n$ and atomic shared variables that support *read*, *write*, and *compare&swap* operations. Each process has five sections of code – Remainder, Recover, Try, CS, and Exit. A *recoverable mutual exclusion algorithm* specifies the code for Recover, Try, and Exit sections of all processes, and the initial values for all local and shared variables. We make no assumptions about the Remainder section and CS other than that none of the shared and local variables of the mutual exclusion algorithm are modified by these sections. All processes are initially in the Remainder section.

2.1 Configuration and step

A *configuration* of the system is specified by the values of all shared variables and the states of the n processes, where the state of a process p is in turn specified by the value of PC_p

(p's program counter) and the values of p's local variables. The configuration changes when a process takes a step. There are two types of steps that a process can take – *normal step* or *crash step* – explained as follows.

- A *normal step by a process p from a configuration C* causes p to perform the instruction that PC_p points to in C. We assume that in a normal step p performs a single operation on a single shared variable, and p's state changes based on the value returned by the shared variable. A special case occurs if p is in the Remainder section: if p is in the Remainder section in a configuration C, a normal step by p from C causes p to transfer control to the Recover section and execute the first instruction of the Recover section. The flow of control from other sections is modeled as usual. If p is in the Try section, after a normal step p either remains in Try or moves to CS. If p is in the CS, after a normal step p moves to the Exit section. If p is in the Exit section, after a normal step p either remains in Exit or moves to the Remainder section. If p is in the Recover section, after a normal step p can be in any of the sections, and the exact rules are specified in Section 2.4.
- A crash step models a process crash. We only model process crashes that occur outside the Remainder section. Specifically, if a process p is in the Recover, Try, CS, or Exit sections in a configuration C, then a *crash step by p from C* sets PC_p to the Remainder section and sets all the local variables of p to arbitrary values. (The crash step does not affect any shared variables since they are assumed to reside in the non-volatile memory, which is unaffected by the crash failures of processes.)

2.2 Execution and Attempt

From the above, we see that a step is determined by which process takes the step and whether the step is normal or crash. Thus, a *step* is an element of $\{1, 2, \ldots, n\} \times \{\text{normal}, \text{crash}\}$. A *schedule* is any finite or infinite sequence of steps. An *execution* corresponding to a schedule $\sigma = s_1, s_2, \ldots$ is $C_0, s_1, C_1, s_2, C_2, \ldots$, where C_0 is the initial configuration specified by the mutual exclusion algorithm, C_1 is the configuration after step s_1, C_2 is the configuration after steps s_1 and s_2, and so on.

Let E be an execution and s be a step by a process p from a configuration C in E. We say *p initiates an attempt in step s* if p is in the Remainder section in C and p's latest step before s is a normal step. We say *p completes an attempt in step s'* if s' is a normal step by p that moves p to the Remainder section. An *attempt* by p in E is a fragment of E that starts with an attempt initiation step s by p and ends with p's earliest attempt completion step s' that follows s. We say *p is active in a configuration C* if C occurs in an attempt by p. It is important to note that p might visit the Remainder section multiple times during an attempt because of its crash steps; thus, p can be active even when it is in the Remainder section.

2.3 Basic properties

Mutual exclusion, bounded exit, and starvation-freedom are properties normally required of any algorithm. The last two properties require suitable adaptation from how they are normally stated for the failure-free setting.

Mutual Exclusion: *An algorithm satisfies Mutual Exclusion* if at most one process is in the CS in every configuration of every execution.

Bounded Exit: The bounded exit property stipulates that a process be able to relinquish its access to the CS without being obstructed by other processes. Formally, *an algorithm satisfies Bounded Exit* if there is an integer b such that, for all executions E and for all

processes p, if p is in the Exit section in any configuration in E and the subsequent steps of p in E are normal steps, then p moves to the Remainder section in at most b of its own steps.

Starvation Freedom: Intuitively, starvation-freedom requires that every process that initiates an attempt eventually enters the CS. However, it is impossible to satisfy this condition unless each process that enters the CS eventually gives up the CS, no process fails infinitely many times in an attempt, and no active process permanently stops taking steps. The last condition means that if a process crashes, it must eventually restart and perform normal steps. We formalize starvation-freedom as follows.

An execution E is *fair* if, for all processes p, we have:

(i) if p is in the CS and does not crash while there, then p subsequently enters the Exit section;

(ii) p has only a finite number of crash steps in any one attempt; and

(iii) if p initiates an attempt then either p completes that attempt or p has an infinite number of normal steps.

An algorithm satisfies Starvation Freedom if in every fair execution every process that initiates an attempt enters the CS in that attempt.

2.4 Well-formedness: how control transfers upon crash and restart

We know from the definition of a normal step that control moves from the Remainder section to the Recover section when a process initiates a new attempt or when a process restarts after a crash during an ongoing attempt. Control moves out of the Recover section either because the process crashes (and the crash step moves the control to Remainder) or because the process eventually completes the Recover section at some normal step. In the latter case, where control moves to is governed by the following expectations:

(i) a process must enter the CS in each of its attempts,

(ii) if a process enters the Recover section because of a crash in the CS, then the Recover section will put the process right back in the CS, and

(iii) if a process enters the Recover section because of a crash in the Exit section, the Recover section can put the process back in the Exit section, bypassing the Try and CS.

More specifically, let s be a normal step by p in which p completes the Recover section, and s' be the latest step by p before s in which p initiates an attempt or p crashes outside of the Recover section (i.e., in Try, CS, or Exit). Then, the rules for where the control moves to after step s are as follows:

- If s' is an attempt initiation step, then s moves control to Try section or CS.
  Informally, if p enters the Recover section from the Remainder section due to an attempt initiation step, when control transfers out of the Recover section eventually due to a normal step, it transfers to Try section or CS.
- If s' is a crash step while p is in Try section, then s moves control to Try section or CS.
  Informally, if p crashes in the Try section and subsequently enters the Recover section, when control transfers out of the Recover section eventually due to a normal step, it transfers to Try section or CS.
- If s' is a crash step while p is in CS, then s moves control to CS.
  Informally, if p crashes in the CS and subsequently enters the Recover section, when control transfers out of the Recover section eventually due to a normal step, it transfers to CS.
- If s' is a crash step while p is in Exit, then s moves control to CS, Exit section, or Remainder section.

Informally, if p crashes in the Exit section and subsequently enters the Recover section, when control transfers out of the Recover section eventually due to a normal step, it transfers to CS, Exit section, or Remainder section.

An execution E is *well-formed* if all of the above rules are met in E. An algorithm satisfies *Well-formedness* if every execution of the algorithm is well-formed.

2.5 Critical Section Reentry

Suppose that a process p crashes while in the CS. Well-formedness guarantees that, once p restarts, Recover section puts p back in the CS. Since the data structures that p was manipulating in the CS could be in an inconsistent state at the time of p's crash, it is important that no other process visits the CS between p's crash in the CS and its subsequent reentry into the CS. Golab and Ramaraju captured this requirement through a property that they called *Critical Section Reentry (CSR)*, stated as follows.

An algorithm satisfies Critical Section Reentry if, for all executions E, if a process p crashes inside the CS, then no other process enters the CS before p reenters the CS.

2.6 Wait-free Recovery

Intuitively, the purpose of the Recover section is to enable a process that is restarting after a crash to repair its state and resume from where it was before the crash. It is desirable if a process can do this recovery without being obstructed by other processes, neither by their relative speeds nor by their crashes. Accordingly, we define:

An algorithm satisfies Wait-free Recovery if there is a bound b such that, for all executions E and for all processes p, if p is in the Recover section in a configuration C of E and the subsequent steps of p in E are normal steps, then p moves out of the Recover section in at most b of its own steps.

Wait-free recovery, together with well-formedness, yields two significant benefits:

1. When a process that crashed in the CS restarts, it gets back into the CS in a bounded number of its own steps, regardless of whether other processes are slow, fast, or have crashed.
2. Similarly, when a process that crashed in the Exit restarts, it gets back into the CS or Exit in a bounded number of its own steps, which means that it can complete the protocol without any obstruction from others.

Furthermore, Wait-free Recovery implies Critical Section Reentry, as shown below.

▶ **Lemma 1.** *Wait-free Recovery, together with Well-Formedness and Mutual Exclusion, implies Critical Section Reentry.*

Proof. Suppose that p crashes inside the CS at some time t. When p subsequently restarts, Well-Formedness and Wait-free Recovery ensure that p will reenter the CS at some time t'. Assume, for a contradiction, that Critical Section Reentry is violated because some process q gets into the CS at some time τ such that $t < \tau < t'$. Let C be the configuration at time τ. In C, q is in the CS and p is in the Remainder or Recover sections. If p alone takes steps from C, then Well-Formedness and Wait-free Recovery ensure that p will enter the CS, thereby violating Mutual Exclusion. ◀

2.7 FCFS

Intuitively, FCFS (First Come First Served) requires that processes enter the CS in the order in which they request the CS [8]. For the traditional model where processes are assumed not to crash, Lamport [8] formalized this intuition by (i) stipulating that the Try section be structured as a bounded section of code, called the *doorway*, followed by the rest of the Try section code that he called the *waiting room*, and (ii) requiring that if a process p is in the waiting room in an attempt A before a process p' initiates an attempt A', then p' does not enter the CS in attempt A' before p enters the CS in attempt A. For the traditional model where processes are assumed not to crash, an equivalent formulation that avoids this syntactic separation of Try into doorway and waiting room would be as follows:

> An algorithm satisfies *FCFS* if there is a bound b such that, for all executions E and for all attempts A and A' in E, if A is an attempt by p, A' is an attempt by p', and p executes b steps in attempt A before p' initiates attempt A', then p' does not enter the CS in attempt A' before p enters the CS in attempt A.

In the current setting, in order to get to the CS, a process needs to execute not only the Try section, but also the Recover section where the attempt is initiated. Therefore, formulating FCFS via doorway and waiting room is cumbersome. Consequently, we have chosen to adapt the above stated alternative specification to the current setting. Two issues arise in this exercise, as we discuss below.

- A process that initiates an attempt might crash before completing the b steps required to establish its "priority". Therefore, to prohibit p' from entering the CS before p, the specification should require that p completes b *consecutive normal steps* before p' initiates its attempt.
- A process might enter and leave the CS multiple times within the same attempt if it crashes repeatedly. In particular, if a process leaves the CS and crashes in the Exit section, subsequent execution of the Recover section can put the process back in the CS. Therefore, the final phrase in the FCFS specification that states "p' does not enter the CS in attempt A' before p enters the CS in attempt A" is revised to "p' does not enter the CS in attempt A' before p *first* enters the CS in attempt A".

Putting these elements together, the final specification of FCFS is as follows. Below, when we say "p performs b consecutive normal steps in attempt A before p' initiates attempt A'" we mean that the sequence of steps that p performs in attempt A (before p' initiates A') includes b consecutive steps all of which are normal.

An algorithm satisfies FCFS if there is a bound b such that, for all executions E and for all attempts A and A' in E, if A is an attempt by p, A' is an attempt by p', and p performs b consecutive normal steps in attempt A before p' initiates attempt A', then p' does not enter the CS in attempt A' before p first enters the CS in attempt A.

2.8 RMR complexity

An operation by a process p on a shared variable X is considered a Remote Memory Reference (RMR) if it involves traversing the processor-memory interconnect. On a Cache-Coherent (CC) machine, a read of X by p counts as an RMR if X was not in p's cache (in which case the read brings X into p's cache), and a non-read operation on X by p always counts as an RMR and removes X from all caches. On a Distributed Shared Memory (DSM) machine, shared memory is partitioned and each process hosts a partition. An operation by p on X, whether a read or a non-read, is counted as remote if and only if X is not in p's partition.

For mutual exclusion algorithms that are designed to run on a CC or a DSM machine, the standard performance metric is the *RMR complexity*, which is the worst case number of RMRs that a process performs in a single attempt. Unlike in a non-recoverable algorithm where a process executes the Try and Exit sections exactly once in an attempt, a process in the current setting can execute the Recover, Try, and Exit sections an unbounded number of times within the same attempt because of its repeated crashes during the attempt. Therefore, we express the RMR complexity of an attempt in terms of n and f, where n is the number of processes for which the algorithm is designed and f is the number of times the process crashes during the attempt.

2.9 Specification of Recoverable Mutual Exclusion

The *recoverable mutual exclusion problem* is to design an algorithm that satisfies Mutual Exclusion, Well-Formedness, Starvation Freedom, Bounded Exit, and Critical Section Reentry. We solve this problem with an algorithm that additionally satisfies FCFS and Wait-free Recovery.

3 Important differences with prior works

Golab and Ramaraju's work [4] is the first to explore the use of non-volatile memory to help processes recover from crash when solving mutual exclusion. The subsequent work by Golab and Hendler [3] uses the same model. Our model and specification differs from these in two significant ways.

- In Golab and Ramaraju's model, the Recover section always puts a process p in Try, thereby causing p to execute Try, CS, and Exit sections in that order each time it restarts after a crash. Consequently, even if the crash occurs after p has completed the CS, when p restarts, it is forced to compete once more for the CS (in the Try section). In contrast, in our model the Recover section can put p in any of the sections. In particular, if p crashes while in the Exit section, when p later restarts, the Recover section will put p directly in CS, Exit, or Remainder, bypassing the needless repetition of Try.
- The properties of wait-free recovery and well-formedness that we have defined differentiate our two works, and have significant implication to how quickly a process can recover after a crash. For instance, suppose that a process p crashes while in the CS. When p subsequently restarts, our well-formedness and wait-free recovery properties together ensure that p will get back into CS in a bounded number of its own steps, i.e., without any waiting whatsoever regardless of however many other processes crashed or are slow. In contrast, Golab and Ramaraju's specification does not have such a guarantee.

We note that, given an algorithm A designed for our model, it is straightforward to transform it into an algorithm A' that conforms with Golab and Ramaraju's model: if the Recover section in A puts a process in Exit section, then the Recover section in A' will make p complete the Exit code (within the Recover section itself) and then send it to Try section.

4 An auxiliary min-array object

Our recoverable FCFS mutual exclusion algorithm, presented in the next section, relies on a special object O that we call a *min-array*. A min-array O has n locations, one per process, and supports two types of operations:

(i) *write*(v), which when executed by process p sets $O[p]$ to v, and
(ii) *findmin*$()$, which returns the minimum value in the array.

No hardware directly supports a min-array, so we implement it using read, write, and CAS operations. Such an implementation of a min-array O specifies code for two procedures, namely, $O.write(p, v)$ that p can execute to set $O[p]$ to v and $O.findmin()$ that a process can execute to get the minimum element in O. For the implementation to be useful in the design of our algorithm, we require it to satisfy wait-freedom and idempotence, which are explained below.

- As always, wait-freedom means that if p invokes $O.write(p, v)$ or $O.findmin()$ and executes the steps of that procedure without crashing, then p will complete and return from that procedure in a bounded number of its own steps [5].
- To explain what we mean by idempotence, let us begin with the notion of a partial execution by p of a procedure Π, where Π is either $O.write(p, v)$ (for a fixed v) or $O.findmin()$. An execution of Π is partial if p invokes Π and performs an arbitrary number of steps of Π, but does not run Π to completion. With most implementations, if p executes Π partially and then reexecutes Π from the start, the implementation can go completely haywire and return arbitrary responses. We however require the implementation to behave gracefully in such a scenario because, if p crashes in the middle of Π, by our model it goes to the Remainder section and subsequently when it starts running again, it might reexecute Π. Motivated by this requirement, we define an *epoch of* Π *by* p as consisting of one or more partial/complete executions of Π. We say the epoch has length f if it consists of f partial/complete executions of Π. By *idempotence* we mean that (i) if the epoch contains at least one complete execution of Π, then the epoch linearizes to some point within the epoch; (ii) if the epoch does not contain a complete execution of Π, then either the epoch never takes effect or it linearizes to some point after the start of the epoch; and (iii) if the same process executes an epoch e of Π and later an epoch e' of a different procedure Π', e' will not linearize before e. (Reader unfamiliar with linearizability is referred to [6].)

Jayanti designed a wait-free, linearizable implementation of a min-array [7] which, although not claimed in his paper, is idempotent. That implementation uses LL/SC operations and has an RMR complexity of $O(\lg n)$ for $O.write(p, v)$ and $O(1)$ for $O.findmin()$. We make a simple modification so that the implementation uses CAS instead of LL/SC and has an RMR complexity of $O(f + \lg n)$ for an epoch of $O.write(p, v)$ of length f. We briefly describe this implementation in the appendix and summarize the result below.

▶ **Lemma 2.** *The algorithm in Algorithm 2 and 3 in the appendix presents a wait-free, idempotent implementation of a min-array O using read, write, and CAS primitives. The RMR complexity of an epoch of O.*write(p, v) *of length f is $O(f + \lg n)$ and the RMR complexity of an epoch of O.*findmin$()$ *of length f is $O(f)$.*

5 The Algorithm

Our Recoverable Mutual Exclusion algorithm is presented in Algorithm 1. We assume that all the shared variables are stored in the Non-volatile Main Memory, and the local (or private) variables are stored in the respective processor registers.

5.1 Shared variables and their purpose

We describe below the role played by each shared variable used in the algorithm.

- Go$[p]$: This is a boolean flag that process p busywaits on, before entering the CS. This variable is set to *true* by p at the start of its Try Section. When a process q makes p the

Algorithm 1 Recoverable Mutual Exclusion Algorithm with FCFS and Wait-free Recovery. Algorithm for process p.

Constants
$\mathcal{P} = \{1, 2, \ldots, n\}$ // Set of process names.
Shared variables (stored in NVMM)
REGISTRY : A min-array, initially empty.
CSOWNER $\in (\mathcal{P} \times \mathcal{P}) \cup (\perp \times \mathcal{P})$, initially $(\perp, 1)$; supports *read*, *write*, and *CAS* operations.
$\forall p \in \mathcal{P}$, GO$[p]$ is a boolean initialized to *true*; supports *read* and *write* operations.
$\forall p \in \mathcal{P}$, REM$[p]$ is a boolean initialized to *true*; supports *read* and *write* operations.
$\forall p \in \mathcal{P}$, MYTOKEN$[p]$ is an integer initialized to ∞; supports *read* and *write* operations.
TOKEN is an integer initialized to 0; supports *read* and *CAS* operations.
Private variables
$\forall p, g_p$ is a boolean arbitrarily initialized, local to p.
$\forall p, t_p$ is an integer arbitrarily initialized, local to p.
$\forall p, i_p$, j_p, s_p, α_p contain a process name from $\mathcal{P}$, all initialized arbitrarily, and local to p.

```
    Recover Section
1.  if REM[p] == true goto Line 7
2.  g_p ← GO[p]
3.  if ((s_p, −) ← CSOWNER) ∧ (s_p ≠⊥)
4.      if ¬GO[s_p]
5.          if (s_p, p) == CSOWNER
                if s_p == p goto Line 16
                else α_p ← s_p; goto Line 23
6.  t_p ← MYTOKEN[p]
    if ¬g_p
        if t_p == ∞ goto Line 9
        else goto Line 12
    else if s_p == p
        if t_p == ∞ goto Line 19
        else goto CRITICAL SECTION
    else goto Line 21

    Try Section
7.  GO[p] ← false
8.  REM[p] ← false
9.  t_p ← TOKEN
10. CAS(TOKEN, t_p, t_p + 1)
11. MYTOKEN[p] ← t_p
12. REGISTRY.write(p, (t_p, p))
13. if ((i_p, j_p) ← CSOWNER) ∧ (i_p ==⊥)
14.     if REGISTRY.findmin() == (t_p, p)
15.         if CAS(CSOWNER, (⊥, j_p), (p, p))
16.             GO[p] ← true
17. wait till GO[p] == true

    Exit Section
18. MYTOKEN[p] ← ∞
19. REGISTRY.write(p, (∞, ∞))
20. CSOWNER ← (⊥, p)
21. if ((−, α_p) ← REGISTRY.findmin()) ∧ (α_p ≠ ∞)
22.     if CAS(CSOWNER, (⊥, p), (α_p, p))
23.         GO[α_p] ← true
24. REM[p] ← true
```

owner of the CS, q releases p from its busywait loop by assigning *false* to GO$[p]$. The operations supported by GO$[p]$ are read and write. To achieve the local-spin property, GO$[p]$ is allocated to p's memory module.

- REM$[p]$: This is a boolean flag that process p uses during recovery to distinguish whether p is active in an attempt or if p entered the Recover section from the Remainder section to initiate a new attempt. This variable is set to *false* by p during its Try Section indicating that it has initiated an attempt. p completes its attempt by setting REM$[p]$ to *true* in the Exit Section. The operations supported by REM$[p]$ are read and write.
- TOKEN is an integer variable supporting read and CAS operations. TOKEN is used to implement a counter so that its values can be used to assign token numbers to processes requesting the CS: in the Try section, a process reads TOKEN to get its token number and then increments TOKEN. As a result, if p executes up to Line-10 and obtains a token during an attempt A before q begins its attempt A', p will get a smaller token number than q (this fact helps achieve the FCFS and starvation-freedom properties).
- REGISTRY is a min-array that supports `write` and `findmin` operations. We require that the two operations satisfy wait freedom and idempotence as mentioned in Section 4. In our algorithm, REGISTRY acts like a queue by holding the names of processes waiting to enter the CS, and orders them according to their token numbers. Specifically, in the Try section, a process p inserts in REGISTRY an element (t, p) (Line 12), where t is p's token

number. When exiting, p deletes this element (Line 19). The elements in REGISTRY are ordered according to their token numbers: $(t, p) < (t', q)$ if $t < t'$ or $t = t' \wedge p < q$. Thus, the `findmin` operation returns (t, p), where p is the process in REGISTRY with the smallest token. If REGISTRY is empty, `findmin` returns the special value (∞, ∞).

- MYTOKEN$[p]$ is an integer variable supporting read and write operations. It is used by processes to remember the token numbers they use while registering their attempt. A process may acquire a token, attempt to insert its entry into the REGISTRY, and crash while completing the insertion. So that it does not acquire a different token and attempt to insert the new token into REGISTRY after recovery from that crash, the process remembers the token that it acquired by recording the token into MYTOKEN$[p]$.
- CSOWNER: This variable holds a tuple both components of which hold process names. The first component holds the name of the process that currently owns the CS. If no process currently owns the CS, the first component holds the value $\perp$. The second component holds the name of the process that gave the ownership of the CS to a certain process. A process may crash just after granting the ownership of CS to some process but right before letting that process into CS, hence not setting the GO flag of that process to *true*. We use the second component for recovery purposes in such a case so that the process can easily remember that it needs to let the waiting process into the CS which is made owner of the CS but not yet let into the CS. The operations supported by CSOWNER are read, write, and CAS.

5.2 Informal description

We now describe informally the algorithm in Algorithm 1. When a process p wants to enter the CS from the Remainder section, it executes an attempt initiation step. In this step it executes the first instruction of Recover and finds that REM$[p]$ is *true*. Therefore, the process proceeds to Line 7 in the Try section. It sets its GO flag (Line 7), and then signals that it is now active in an attempt by setting REM$[p]$ to *false* (Line 8). It then proceeds to obtain a token for itself and increments the global counter (Line 9-10). It also saves the token it obtained for itself (Line 11), so that in the event of a crash it does not obtain a different token and try registering its attempt with the new token. Then, it inserts its name, tagged with its token, into the REGISTRY (Line 12). If a process p executes normal steps upto Line 12 in attempt A before q initiates an attempt A', then q does not enter the CS in A' before p first enters the CS in A. After executing the Line 12, p tries to capture the CS on its own by first confirming that the CS is free to be occupied (Line 13), the process with the smallest token in the REGISTRY is p itself (Line 14), and then attempting to capture the CS for itself (Line 15). If p succeeds in capturing the CS, it sets its own GO flag to *true* (Line 16), informing itself that it no longer needs to wait. Following this, p busy-waits until it is informed that it no longer needs to wait (Line 17). It then enters the CS. When p leaves the CS, it first wipes out the token that it obtained for the attempt (Line 18). It then removes its own name from the REGISTRY (Line 19), and marks the CS as available (Line 20). Following this, p tries to capture the CS for a waiting process (Line 21-22) and informs the waiting process that it no longer needs to wait (Line 23), if successful in capturing the CS. Whether p lets another process into the CS or not, it completes the attempt by setting REM$[p]$ to *true* (Line 24) to indicate that it is no longer active in an attempt.

Next we describe the Recover section. A process p may go to the Recover section due to one of two reasons:

(i) p may crash,

(ii) p may initiate a new attempt due to an attempt initiation step.

Therefore, p first reads Rem[p] to determine if it wants to initiate an attempt (Line 1). If so, p proceeds to the Try Section. Otherwise, p reads its own Go flag so that it can decide later where it crashed during the attempt (Line 2). p then checks if any process is currently the owner of CS. The section of code on Lines 4-5 is executed when p learns that some process owns the CS. Suppose p learns that some process owns the CS. Then p checks the Go flag of that process to determine if that process was informed to no longer wait. If that process is still waiting without any information, p checks if the CS is still owned by the same process, and whether p installed that process as the owner. If that is so, p was either installing itself as the owner, in which case it has to go back to the Try section (i.e. Line 16), or while exiting p was installing some other process as the owner of the CS, in which case it has to go to the Exit section again. If any of the checks at Lines 4-5 fail, p is not responsible in informing any waiting process. If no process is the owner, then p definitely did not crash while letting some process into the CS. Therefore, p proceeds to read its own token (Line 6). Depending on p's token value, Go flag, and the value of s_p (used only to identify if p itself is the owner of CS), p decides whether to go back to Try, CS, or Exit section.

5.3 The subtle features of the algorithm

From the above description it might not be clear why we perform certain operations in the algorithm, namely,

(A) the need to set Rem[p] after setting Go[p] (Line 7-8),

(B) the need to read CSOwner twice in the Recover section (Lines 3, 5).

We demonstrate below that the first feature is necessary to preserve Well-formedness. The second feature is necessary to preserve Mutual Exclusion.

The need for feature A: In order to distinguish between a fresh attempt and a recovery to an active attempt, we have to maintain the boolean variable Rem[p]. The placement of the lines that modify Rem[p] is also important. Suppose, instead, we set Rem[p] to *false* in the first line of the Try Section. Then it is possible for a process to re-initiate an already completed attempt as we demonstrate in the following scenario. Process p initiates a new attempt, obtains the lock and executes the CS, and continues to the Exit section. p executes all the steps upto Line 24. Right when $PC_p = 24$, p executes a crash step. The next steps that p executes are from the Recover section. p notices that Rem[p] = *true*, MyToken[p] = ∞, and Go[p] = *true*. Since p didn't crash at Line 23 (which it knows after reading CSOwner), it incorrectly concludes that it crashed somewhere in the Try section before obtaining a token for itself. Therefore, p goes back to the Try section instead of the Exit section. This violates Well-formedness. This problem would not arise with p if Rem[p] is set to *false* in Line 8, because by reading Go[p] it can infer that the value of Go[p] is *true* due to its own completed attempt. If p had not initiated an attempt, Go[p] would still be *true*, but Rem[p] would also be *true*. Therefore, p can infer correctly that it needs to go to the Exit instead of Try section.

The need for feature B: Suppose p crashes either at Line 16 or Line 23, i.e., while setting the Go flag of some process. Then, during recovery, in order to conclude that p crashed just before writing to the Go flag, p has to read CSOwner and suppose the second component contains p's name, then it has to read the Go flag of the process whose name appears in the first component of CSOwner. Let s_1 be the step when p reads CSOwner the first time for recovery. Let q be the process whose name p read in the first component of CSOwner

and p read its own name in the second component. Let s_2 be the step when p reads GO$[q]$. Suppose p finds that GO$[q]$ is *false* during s_2. There are two cases applicable here:
(1) During s_1 GO$[q]$ was *true*,
(2) During s_1 GO$[q]$ was *false*.
In the first case, p had already let q into the CS. Between steps s_1 and s_2, q completed that attempt (this would change the value of CSOWNER to something other than (q, p)), and later came back for another attempt setting GO$[q]$ back to *false*. Hence, p found GO$[q]$ as *false* by executing s_2 in the first case. In the second case, q was actually somewhere in the Try during s_1 and s_2 and it continues to be so, therefore, q has to be let into the CS by p. In order to distinguish between the two cases, we read CSOWNER again at Line 5. If p finds that CSOWNER is still (q, p), it concludes that it crashed just before setting GO$[q]$ to *true*. Hence, it goes back to the appropriate line. Otherwise, p hasn't obstructed any process from going into the CS, therefore, it resumes with the rest of the recovery. If p didn't read CSOWNER for the second time in Line 5, p would incorrectly inform q to proceed to CS in the first case discussed above. Hence, q would continue to CS when some other process already occupies it, violating Mutual Exclusion.

5.4 Main theorem

The theorem below presents our main result. We will soon release a Dartmouth College Technical report that includes a detailed proof of correctness based on an invariant satisfied by the algorithm.

▶ **Theorem 3.** *The algorithm in Algorithm 1 solves the recoverable mutual exclusion problem for n processes and additionally satisfies FCFS and Wait-free Recovery. The algorithm uses read, write and CAS operations. On both CC and DSM machines, a process that fails at most f times in an attempt performs at most $O(f + \lg n)$ RMRs.*

Acknowledgment. We are grateful to the anonymous reviewers for their careful and detailed reviews and Wojciech Golab for helpful discussions.

References

1 Hagit Attiya, Danny Hendler, and Philipp Woelfel. Tight RMR Lower Bounds for Mutual Exclusion and Other Problems. In *Proc. of the Fortieth ACM Symposium on Theory of Computing*, STOC'08, pages 217–226, New York, NY, USA, 2008. ACM.
2 E. W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Commun. ACM*, 8(9):569–, September 1965.
3 Wojciech Golab and Danny Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC'17, pages 211–220, New York, NY, USA, 2017. ACM.
4 Wojciech Golab and Aditya Ramaraju. Recoverable Mutual Exclusion: [Extended Abstract]. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC'16, pages 65–74, New York, NY, USA, 2016. ACM.
5 Maurice Herlihy. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.
6 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.
7 Prasad Jayanti. f-arrays: Implementation and Applications. In *Proceedings of the Twenty-first Symposium on Principles of Distributed Computing*, PODC'02, pages 270–279, New York, NY, USA, 2002. ACM.

8 Leslie Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Commun. ACM*, 17(8):453–455, August 1974.

9 John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

10 Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 401–410, New York, NY, USA, 2012. ACM.

11 Aditya Ramaraju. RGLock: Recoverable mutual exclusion for non-volatile main memory systems. Master's thesis, University of Waterloo, 2015. URL: `https://uwspace.uwaterloo.ca/handle/10012/9473`.

12 Jae-Heon Yang and Jams H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, 1995.

A Min-Array implementation with f-arrays

In this section we describe Jayanti's f-arrays implementation [7]. Detailed explanation and a proof of correctness are presented in Jayanti's paper. Here we present a modified version of the implementation so that it incurs only a constant extra RMR in the event of a failure during any procedure call. Since the original implementation uses LL/SC, we give an equivalent implementation that uses CAS and unbounded counters.

Algorithm 2 gives the tree-based implementation of the f-arrays using CAS. In the implementation there is one node for every process, for a total of n nodes, representing the array cells. These n nodes are the leaves of a binary tree such that the height of the tree is $\lg n$.

In computing the min function, the idea is that a process enters an element in its own cell first. It then traces a leaf to root path so that at each node in the path it takes the values of that node's children and writes the minimum among these values into the node. This continues until the process reaches the root, where it stores the minimum value it encountered in the path.

The shared variable SavedState is an additional component that stores the address of a node in the tree. During the execution of the `write` operation a process may abort (in our case crash) to resume the execution at a later point of time. When a process invokes a procedure call to `write`, we first check if the call was to write a different value from the previous call (Line 26). If so, then start percolating the value into the f-array from the leaf nodes of the tree as explained above. Otherwise, the current call is an invocation to write the same value as before, hence, to maintain idempotence, start percolating the value up the tree from the point where the process stopped in the last invocation before the abort.

As described in [7], after the second call to `refresh` we are sure that the value in *val* is percolated upto the node in the tree pointed to by *currNode*. Therefore, we save this progress as a checkpoint along with the value in the shared variable SavedState, and continue percolating the value up the tree.

In each call the function min() (or $f()$ in case of an f-array) in Line 42 takes the minimum value stored in the child nodes of the node pointed by *currNode* and writes the minimum value into *currNode*. The call to refresh happens repeatedly so that a value that is first written to a leaf node in the tree (Line 25) gets percolated up and ultimately once the process reaches the root of the tree, it writes in the root the minimum value in the array. To read the minimum value, the process simply reads the root node and returns the value (Line 37, 38).

Algorithm 2 Tree algorithm to implement f-arrays object with CAS. Algorithm for process p. Adaptation of algorithm is from Figure 4 in [7]. Note, here we explicitly substitute the min() function in Line 42 instead of the function $f()$ since we know we will use this as a min-array. Algorithm 1 does not constrain the structure of the object tree τ'. Therefore, we assume that τ' is a binary tree of height $O(\lg n)$.

Shared objects (stored in NVMM)
τ': Object tree corresponding to a type tree τ as described in Section 4.2 of [7].
$\forall p, L_p$ is the pth leaf of τ'.
Shared variables (stored in NVMM)
$\forall p$, SavedState$[p]$: An array storing the node in the tree to start update from, and the value on which $\texttt{write}_p$ was called. Initialized to $(\&L_p, \infty)$.
Private variables
val, val' are integers, initialized arbitrarily.
$currNode$ is a reference to a node in τ', initialized arbitrarily.
$res, seq, v, v_1, v_2, \ldots, v_k$ are integers, initialized arbitrarily.

procedure $\texttt{write}_p(val)$
25. $L_p \leftarrow val$; $res \leftarrow val$
26. **if** $(((currNode, val') \leftarrow$ SavedState$[p]) \wedge$ $val \neq val')$
27. $currNode \leftarrow L_p$
28. **if** $currNode == root(\tau')$
29. **return** res
30. **repeat**
31. $currNode \leftarrow parent(currNode)$
32. **if** $\neg\texttt{refresh}(currNode)$
33. $\texttt{refresh}(currNode)$
34. SavedState$[p] \leftarrow (currNode, val)$
35. **until** $currNode == root(\tau')$
36. **return** res

procedure $\texttt{read}()$
37. $(v, -) \leftarrow \text{read}(root(\tau'))$
38. **return** v

procedure $\texttt{refresh}(currNode)$
Let $C_1, \ldots, C_k$ be $currNode$'s children
39. $(val, seq) \leftarrow \text{read}(currNode)$
40. **for** $i \leftarrow 1$ **to** k
41. $v_i \leftarrow \text{read}(C_i)$
42. **return** CAS$(currNode, (val, seq),$ $(f(v_1, v_2, \ldots, v_k), seq + 1))$

Algorithm 3 Implementation of min-array using f-arrays (see Algorithm 2; substitute min() function for the function $f()$ in Line 42) adapted from algorithm of Figure 8 in [7]. Algorithm for process p.

Shared variables and objects (stored in NVMM)
$\mathcal{A}$: An f-array of n components as described in Algorithm 2.
Private variables
val, v are integers, initialized arbitrarily.

procedure $O.\texttt{write}(p, val)$
43. $\texttt{write}_p(\mathcal{A}, val)$

procedure $O.\texttt{findmin}()$
44. $v \leftarrow \texttt{read}(\mathcal{A})$
45. **return** v

In Algorithm 3 we give an f-array implementation of Registry used in our algorithm. We adapt this from Jayanti's f-arrays paper [7] except that we do not utilize adaptivity as done in the paper.